



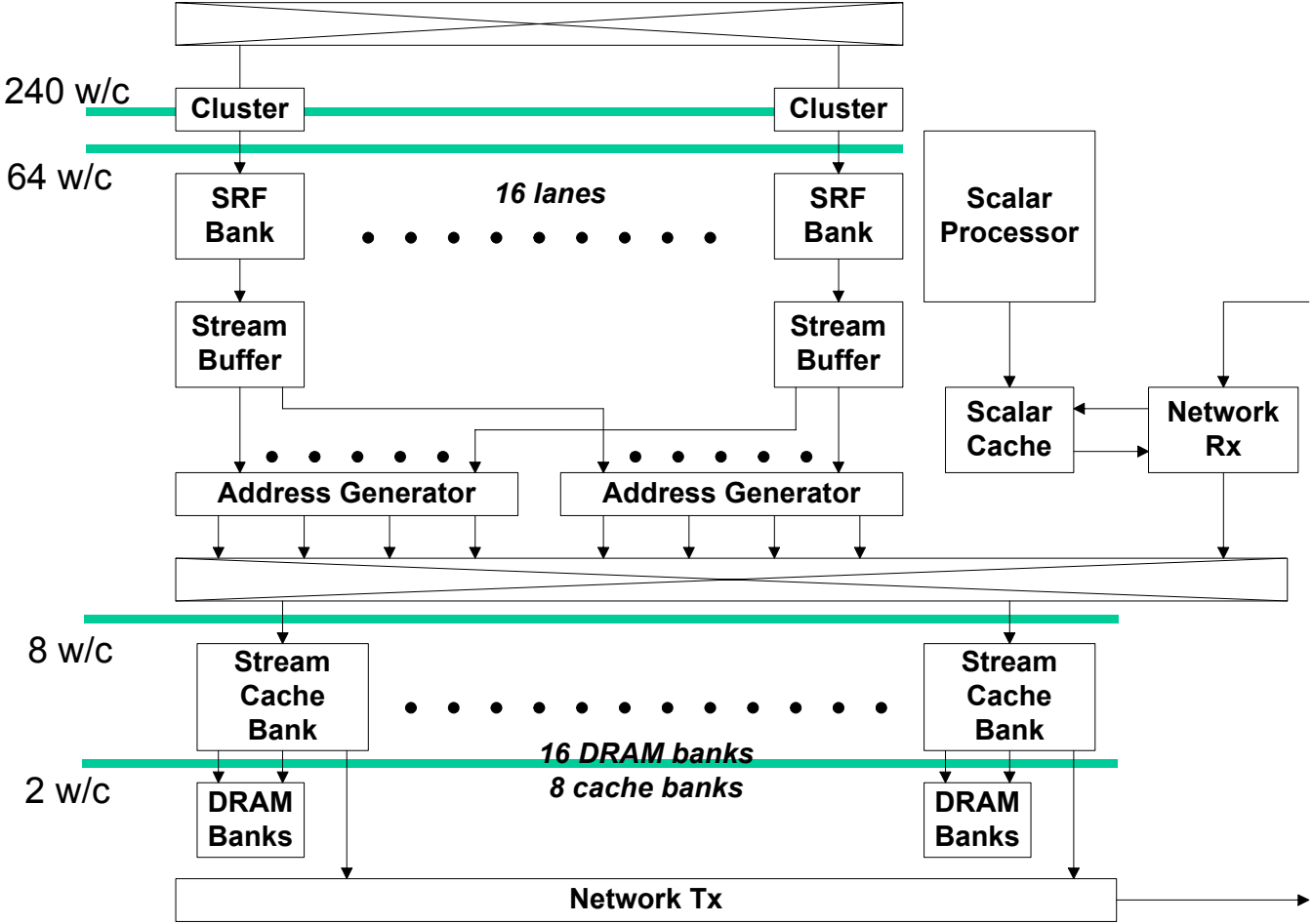
# SSS Memory System Architecture

Timothy Knight  
Jung Ho Ahn  
Bill Dally

# Introduction

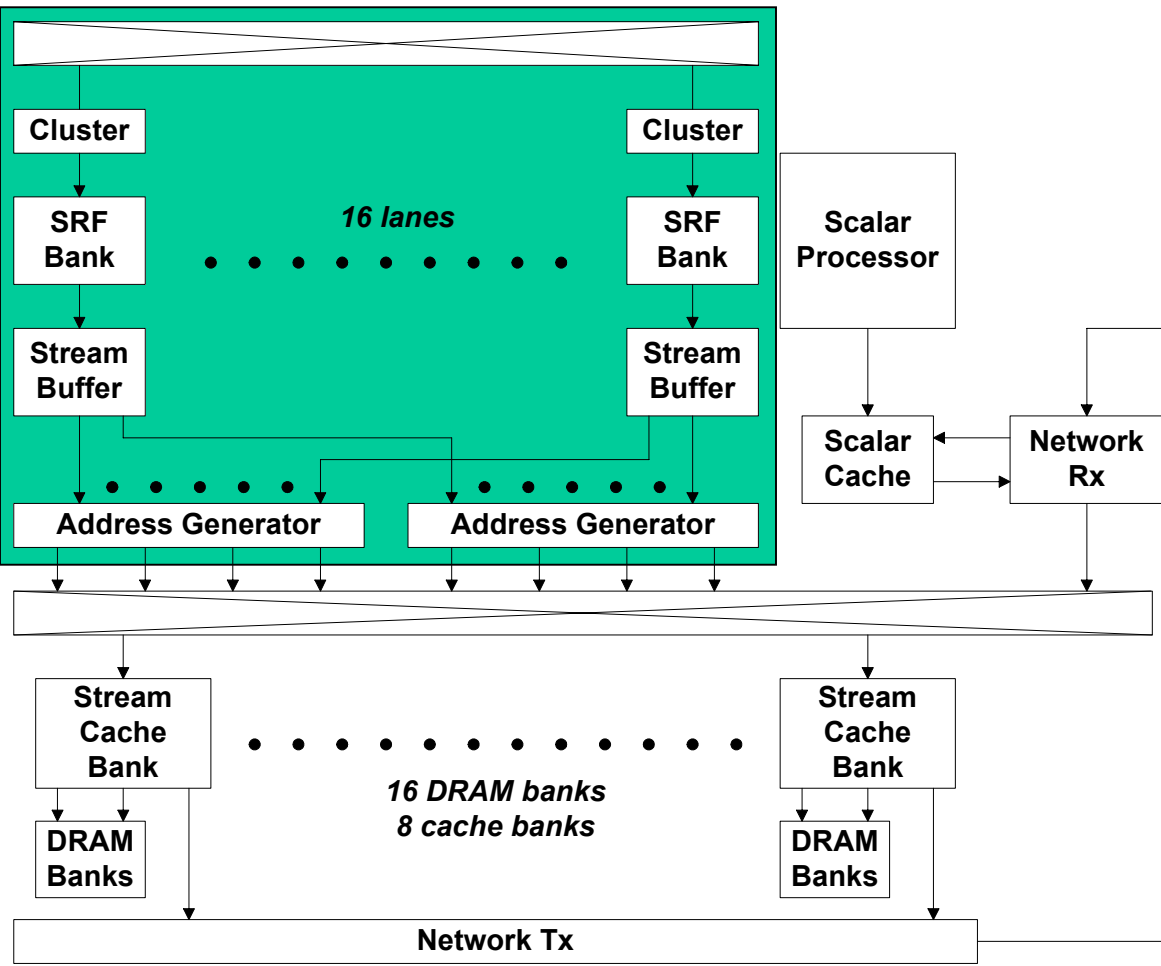
- This presentation will discuss:
  - SSS memory system architecture
    - High bandwidth → banks, lanes
  - Programmer's model of SSS memory system

# SSS Memory System Arch. (1)



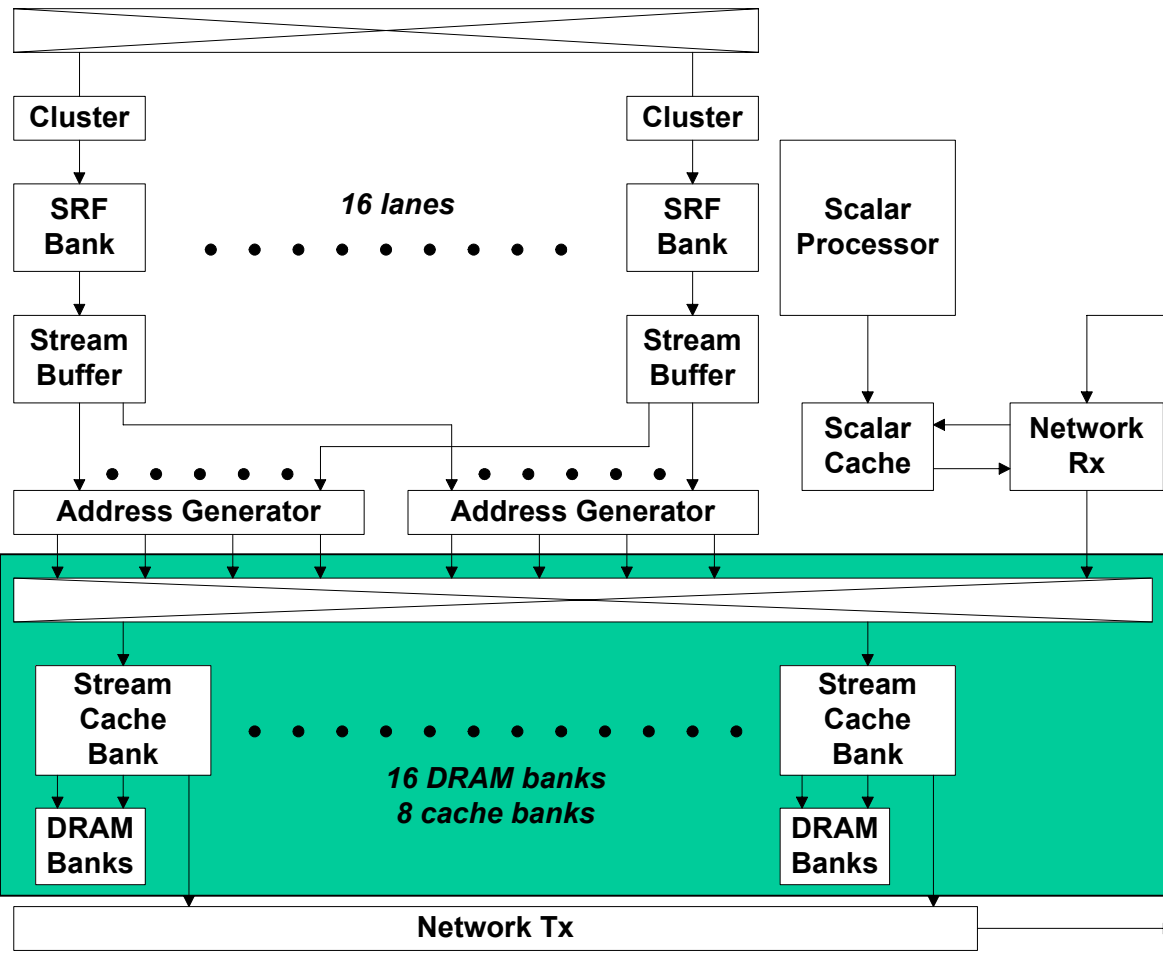
- Requests side
- Bandwidth hierarchy

# SSS Memory System Arch. (2)

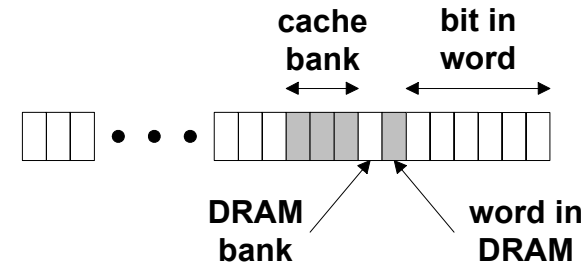


- Stream unit
  - 16 clusters
  - Dedicated SRF bank per cluster
  - AGs issue 8 requests per cycle
  - Inter-cluster 16x16 crossbar

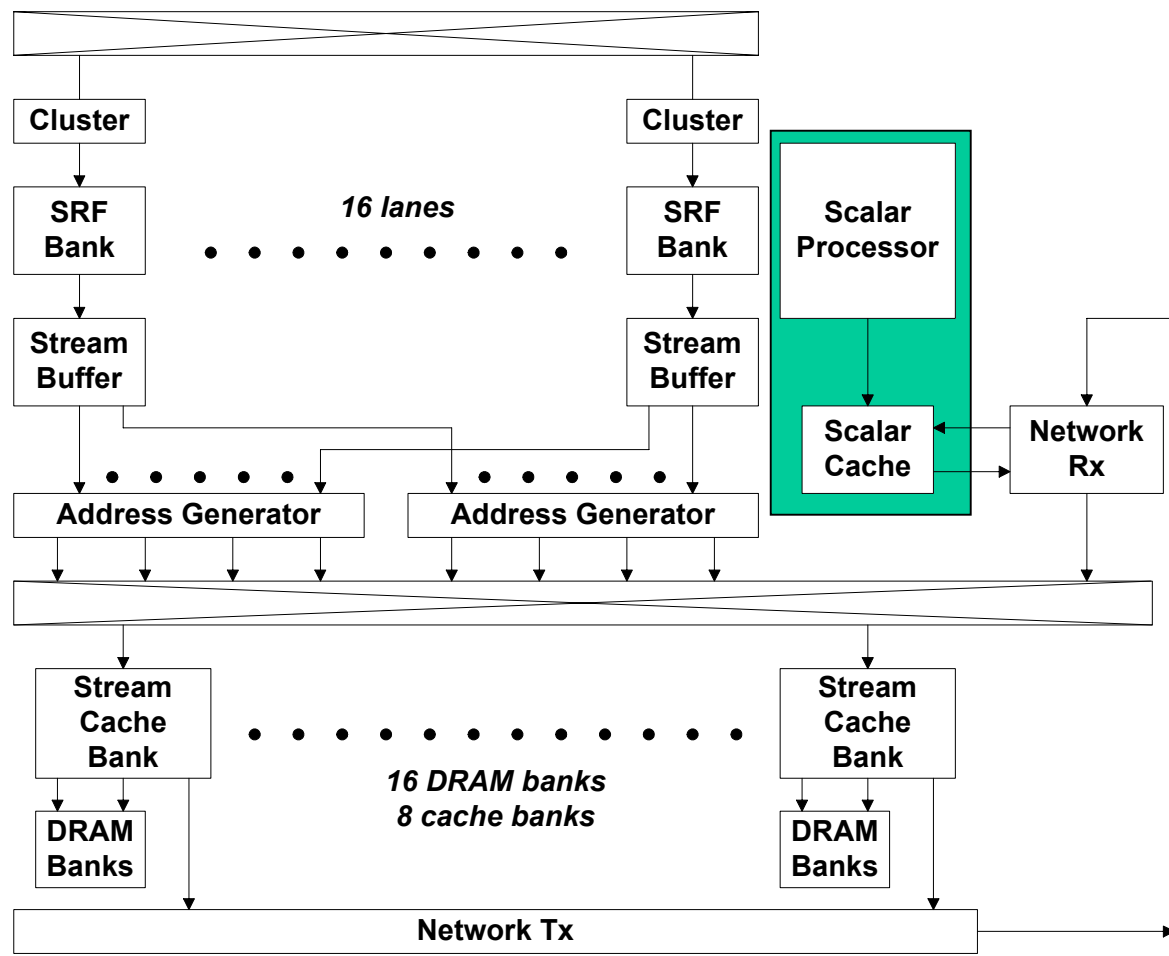
# SSS Memory System Arch. (3)



- Memory system
  - Multi-bank for high throughput
  - Stream cache bank on each pair of DRAM banks
  - Bank selected by address:



# SSS Memory System Arch. (4)



- Scalar unit
  - Scalar processor
  - Scalar cache
    - Snoops requests received over network

# Programmer's Model

- Segments
- Memory consistency
- Cache Coherence
- Scatter-accumulate
- Synchronization

# Segments (1)

- Defined by segment registers in each node
  - 32 registers?
- Can span multiple nodes
  - Interleaved in configurable amounts
- Used for address translation:
  - VA: (segment number, segment offset)
  - PA: (node number, node offset)

# Segments (2)

- Are in one of the following states:
  - Non-cacheable
  - Read-only (cacheable anywhere)
  - Locally scalar-cacheable
  - Locally stream-cacheable
  - Globally stream-cacheable
- Segment state can change at run-time
  - Caches support segment flush via multiple valid bits



# Memory Consistency

- SSS memory system has relaxed ordering
  - Explicit memory barriers (scalar instructions)
    - Per-segment
  - No ordering defined for reads / writes between memory barriers
- Completion of stream load / store / flush instruction is a guarantee of completion of all underlying memory accesses
  - Stream controller handles dependencies

# Memory State Bits

- Each memory word has state bits:
  - Valid
  - Shared read-only
  - Lock-exclusive
  - Lock-accumulate
- Violation raises memory error
  - (e.g.) Reading from a word which was locked-exclusive by another node
  - Provides safety against program errors

# Stream Cache Motivation

- Streams already overcome high latency
  - Amortize it over many memory accesses outstanding simultaneously
- Stream cache acts as a bandwidth amplifier
  - Cache bandwidth is 4x memory bandwidth
- Exploit temporal locality
  - Irregular stream applications

# Stream Cache Architecture (1)

- Stream cache is:
  - Physically addressed
  - Write-back
  - Banked for high throughput
- Line size = 1 word

# Stream Cache Architecture (2)

- Stream instructions explicitly
  - Specify whether the cache may be used on loads and stores
  - Flush, invalidate
  - “Clean” – write back dirty data but don’t invalidate

# Stream Cache Coherence

- Stream caches are not coherent but are safe; they hold only:
  - Read-only data
    - R/O segment or locked-R/O words
  - Private write-back data
    - Locked-exclusive words
    - Guaranteed no other thread will write it
  - Scatter-accumulate data
    - Locked-accumulate words
    - Guaranteed no other thread will clobber partially accumulated value in memory

# Stream Scatter-Accumulate (1)

- The SSS memory system supports “add-and-store” requests
  - Stream of partial sums in SRF are added to memory locations
  - Each addition operation is atomic
  - 64-bit floating point addition is supported
- Ordering of additions is undefined
  - May be a problem with finite precision (64-bit) floating point arithmetic

# Stream Scatter-Accumulate (2)

- Stream unit can issue add-and-store requests to addresses in
  - Non-cacheable segments
  - Stream-cacheable segments
- Add-and-store requests can be processed by
  - Stream cache
  - Memory

# Stream Scatter-Accumulate (3)

- Stream cache entries can be
  - Dirty
    - A load/store cache miss or a pre-fetch caused this entry to be allocated
  - Accumulate-dirty
    - An add-and-store cache miss caused this entry to be allocated

# Stream Scatter-Accumulate (4)

- When an accumulate-dirty entry is flushed or evicted, it is sent to memory as an add-and-store request rather than a store request
  - Enables programs to accumulate partial sums in local stream cache
    - In parallel (on multiple nodes) without synchronization
  - Partial sums in different caches are combined in memory when the caches are flushed

# Stream Fetch-and-Add

- Fetch-and-add stream requests are also supported by memory system
  - Same as add-and-store, but return the original memory value as well as adding it
- Useful for (e.g.) stream stores to the tail pointer of a data structure shared across nodes
  - Atomic manipulations of tail pointer done in hardware with no software synchronization

# Scalar Cache Architecture

- Scalar cache is:
  - Physically addressed
  - Write back
  
- Can supply data to:
  - Scalar processor
  - Network interface

# Scalar Cache Coherence

- Scalar cache can only cache:
  - Local addresses from scalar-cacheable segments
  - Local / remote addresses from read-only segments
- Cache coherent across entire system
  - Snoop requests received over network
- Stream unit requesting data from scalar segment:
  - Performance hit due to bandwidth mismatch
  - Request from AG goes to network Tx, looped back to Rx in case of local address

# Synchronization

- Scalar synchronization instructions:
  - Load-locked / store-conditional (LL/SC)
    - Enable atomic scalar read-modify-write
  - Memory barrier
  - Fetch-and-add (non-cacheable)
- “Software Queuing Lock”
  - Algorithm for multiple nodes trying to acquire a global lock to only spin on a local (cached) variable

# Example: Gridded StreamMD (1)

- Each node contains multiple adjacent grids
- Each grid contains 2 streams of particles
  - Current:
    - Streamed into kernels in this iteration
    - Final kernel outputs are updated particles which are stored in a new grid
  - Next
    - Stream containing particles which have been re-gridded to this grid this iteration; becomes the “current” stream next iteration

# Example: Gridded StreamMD (2)

- Grid data structure (tail pointers of streams, ...)
  - Stream-cacheable during iteration
    - Scattered re-gridded particles written (using fetch-and-add) to “next” tail pointer
  - Scalar-cacheable at end of iteration
    - Tail pointer management done by scalar processor
    - Stream cache flushes and memory barrier performed before segment state changed to scalar-cacheable
- Particle data
  - Stream-cacheable always